二进制漏洞挖掘系列课程-(2)利用 SEH 绕过 GS 安全机制

Writenby 東

实验环境:Win7 sp1 x64
实验工具:vs2013 ImmunityDebug mona.py

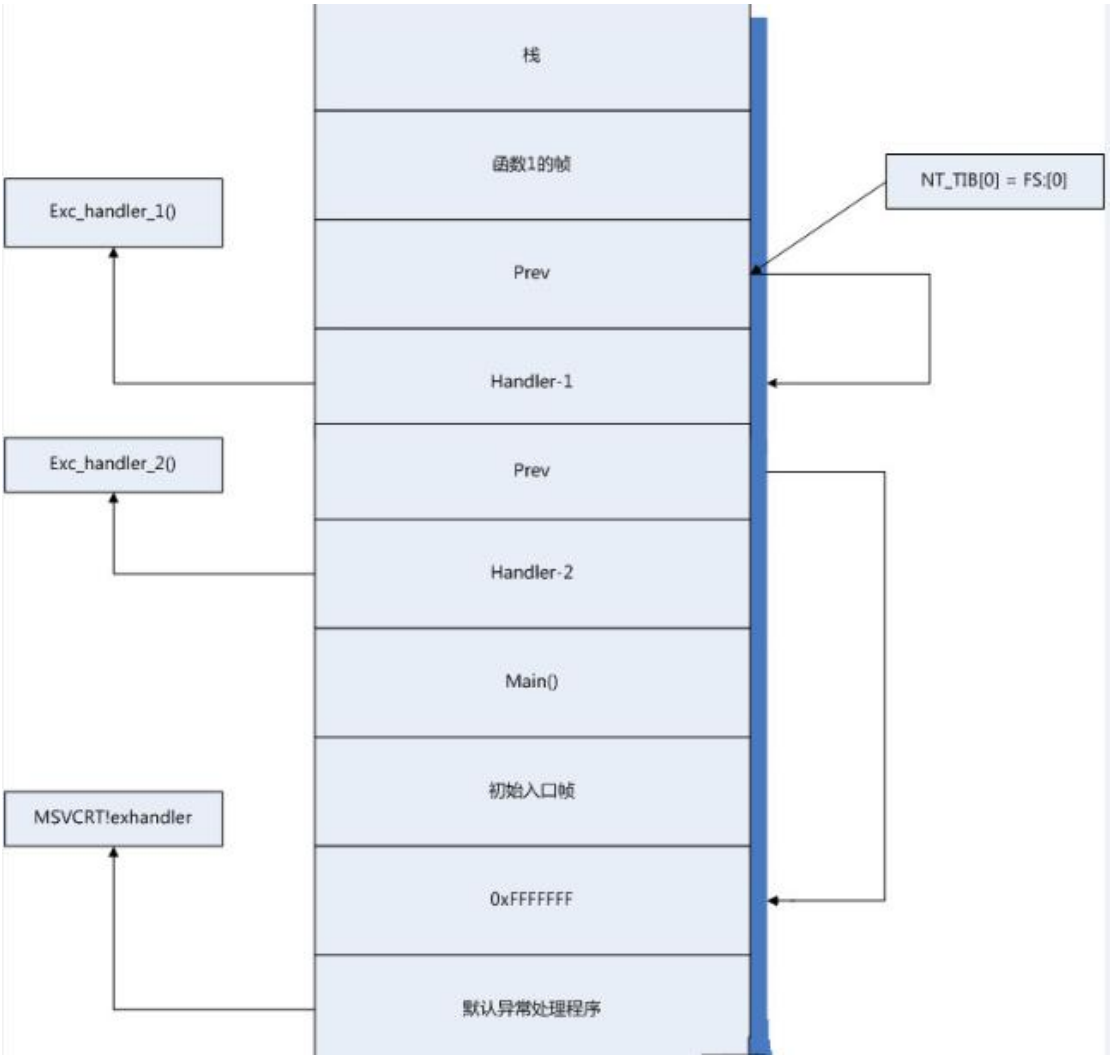
这个专题需要了解两个概念,一个是异常处理机制(seh),还有一个就是 Windows GS 保护.

SEH:

（Structured Exception Hadnling）结构化异常处理
(windows)操作系统提供给程序设计者的强有力的处理程序错误或异常的武器
C 语言中通常通过_try catch 来实现

```
int main()
{
      _try{
                  //可能出现异常崩溃的代码
            }
     _except(EXCEPTION_EXECUTE_HANDLER) {
                  //异常处理程序
            }
     return 0;
}
```

其实 Windows 在原始的程序栈前面添加了一个异常处理结构，该结构由一系列的异常处理链表组成，这条链表的起始点总是放在 TIB（Thread Information Block）的第一个成员中，在 x86 计算机中存储在 FS:[0]寄存器中。链表的最后总是默认处理程序，这个默认处理程序的指针总是 0xFFFFFFFF



GS 保护机制：

Windows 在 **VS7.0(Visual Studio 2003)**及以后版本的 **VisualStudio** 中默认启动了一个安全编译选项——**GS**（针对缓冲区溢出时覆盖函数返回地址这一特征）

GS 保护机制是在函数即将调用的时候向栈桢压入一个 **DWORD** 的随机值,同时也向**.data** 段中存放一个 **Security Cookies**,

1. 被压入栈中的随机值位于 EBP 之前. 在. data 段中的数据实现栈 Cookies 的校验
2. 在函数返回之前，系统将会执行一个额外的安全验证操作，被称作 SecurityCheck
3. Security 当校验发现栈 Cookies 和 .data 的副本不吻合则表明发生溢出
4. 当检测到栈中发生溢出时,系统接管异常,函数不会被正常返回，ret 指令也不会被执行
5. 当栈中发生溢出时，Security Cookie 将被首先淹没，之后才是 EBP 和返回地址

GS 保护机制的实现细节是

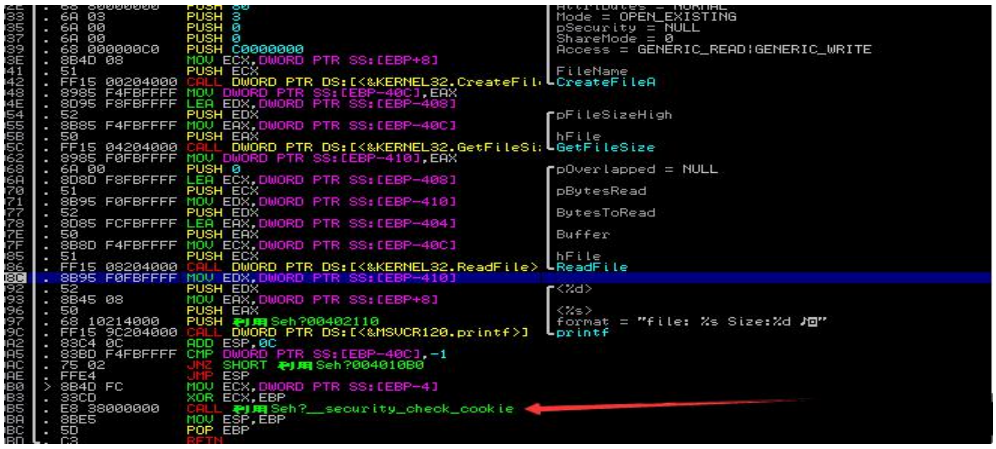
1　系统以.data 段的第一个 DWORD　作为 Cookie 的种子
2　每次程序运行时的 Cookie 的种子都不一样,随机性很强
3　栈桢初始化完毕后用 EBP 异或种子,作为当前函数的 Cookie,以此区别不同函数,增强 Cookie 的随机性
4　　在函数返回前,用 EBP 异或还原出 Cookie 种子

绕过 GS 安全保护的方案

1）　通过覆盖 SEH 链表来阻止系统接管异常处理.
2）　通过改写 C++虚表指针来控制程序流程
3）　用一些未开启 GS 安全保护的函数进行溢出(可能是关键字保护)||小于四字节的 Buf

今天这个课程我们讲通过覆盖 SEH 链表来进行 exploit

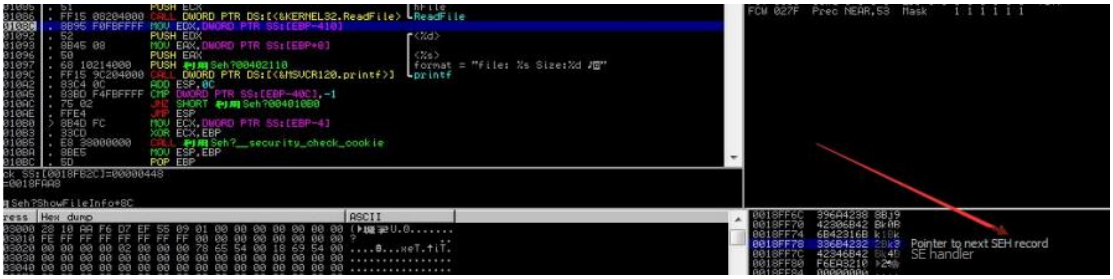
还是上节课的例子,我们开始调试在反汇编窗口查看 ShowFileInfo　这次为了演示 seh 我把 print 函数放在了 ReadFile 下面我们这次看到了在 ShowFileInfo 中 Printf 函数下面有一个 Security_Check_Cookie　这就是我们的 GS 缓冲区检测机制的这个函数,工程项目是 realse 版本的,在项目属性只开启 GS。关闭 优化选项,dep,aslr,safeseh(vs 项目属性选择配置属性->链接器->命令行填写“/SAFESEH:NO　”)



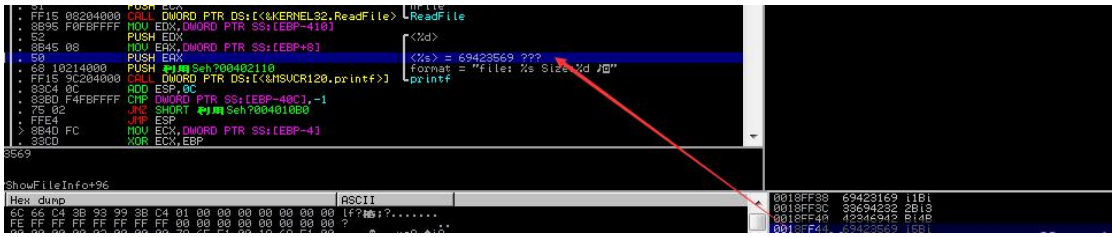
我们可以试试如果和上次一样覆盖掉返回地址当执行到 **Security_Check_Cookie** 的时候,他会检查栈 **Cookies** 和 **.data** 的副本,这时候 **GS** 就分发系统异常处理请求然后就由系统接管处理你这个异常 我们可以先用 mona 插件查看程序当前 seh 链表

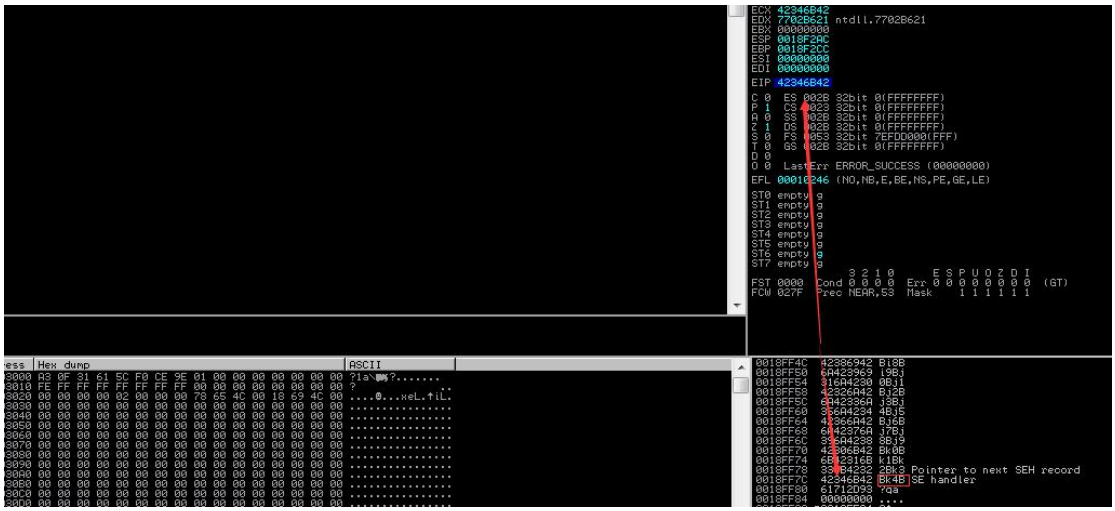
这个地址指向的就是 Pointer to next SEHrecord　下面的 SE hander 是 ntdll 中的系统接管处理。

我们现在的思路就是覆盖掉这个 SEH 异常处理链表 SEH handler　还需要 18*4 个字节才能覆盖掉我们用 mona 生成一个 1024+72 字节的字符串.命令行参数重新载入不要忘了 Imdebug 加参数启动,现在可以看到已经覆盖掉了 **SEH handler** 了

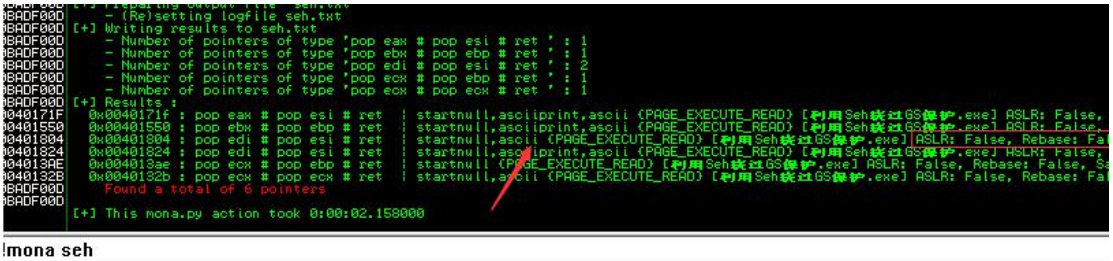


我们继续单步到 **Printf**　看到了参数已经被我们覆盖掉了





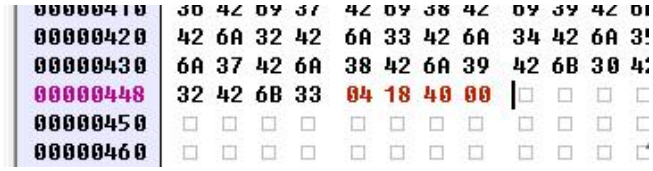
接下来就是寻找个跳板了,seh 通常利用的是 pop pop ret　一旦进入异常处理,就会把 Pointer to next SEH 的这个地址压入栈中进行系统处理,通过 pop pop 然后这个地址 ret 到我们的 eip 中,因为 Pointer to Next..是可控的所以我们控制这个地址来控制 eip,然后就是可以通过 mona 来找 pop pop ret 来覆盖 Se handler



!mona seh

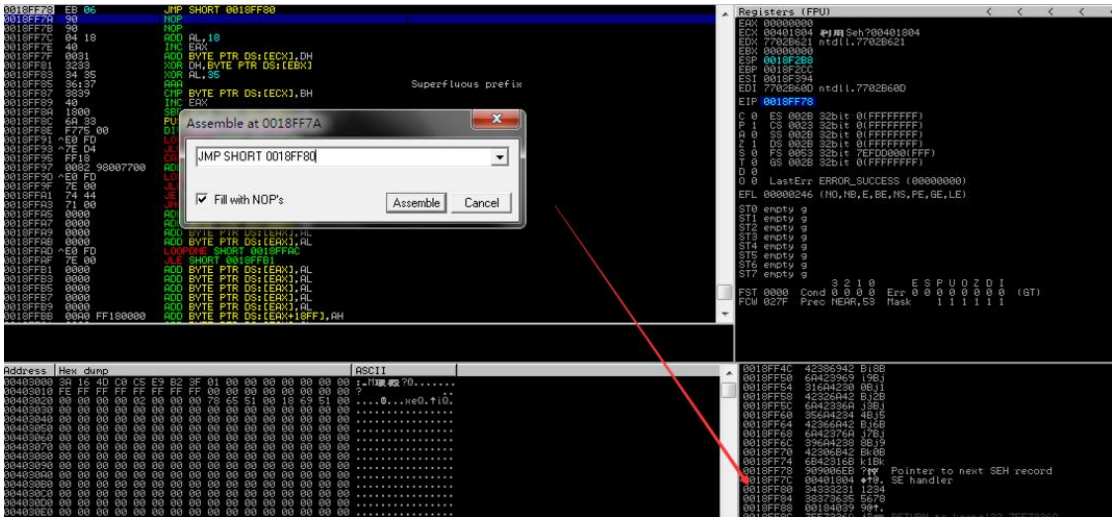
找到这一行 PAGE_EXECUTE_READ，后面的保护机制都是 false

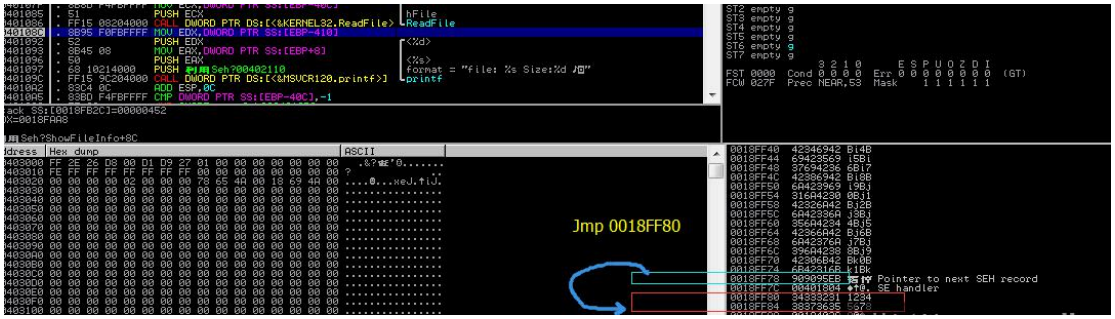
Address=00401804　Message= 0x00401804 : pop edi # pop esi # ret | startnull,ascii {PAGE_EXECUTE_READ} [利用 Seh 绕过 GS 保护.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0-(C:\Users\Administrator\Documents\Visual Studio 2013\Projects\利用 Seh 绕过 GS 保护\Release\利?找到一个 ppr 的地址我们打开十六进制编辑器以小端存储将尾部四个字节覆盖掉



我们已经知道了当执行完我们的 pop popret 会把上一个函数的地址弹到 eip 中我们这个地址也是我们可以控制的. 我们在 Pointer to next SEH record 地址做一个跳板跳转至我们的 shellcode 中但是当前地址已经离栈底很近了,所以我们要把 shellcode 放置在我们的 buf 中,向上跳。但是这里还有一个细节就是当你向上跳转的时候是 jmp 一个负的地址 那么这条 jmp XXXX 这条指令就会撑爆当前的这四个字节空间,覆盖掉了后面的 se handler 数据,所以我们要先在下面找到一个比较近的一块空区域然后在那块区域的地址上写上我们 jmp shellcode 的指令.我们选择 0018FF80 这个地址

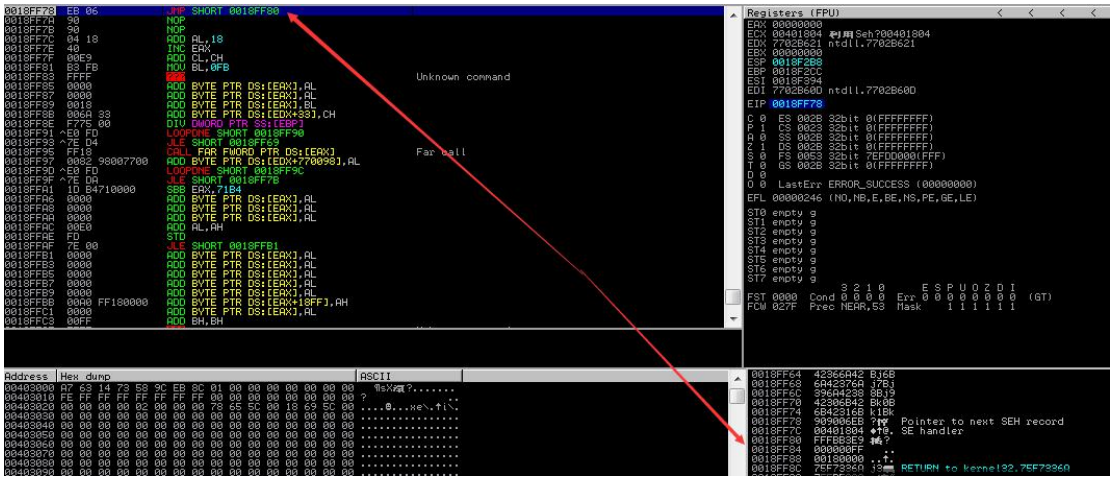
现在我们加长我们的文本,可以看到下图中, 现在搜索我们的 pop pop ret 指令,反汇编窗口 ctrl+G 输入 00401804 在然后再 pop 上下断点,shitf+f9 运行观察 eip,当执行完 ret 指令后的当前指令修改为 jmp 0018FF80



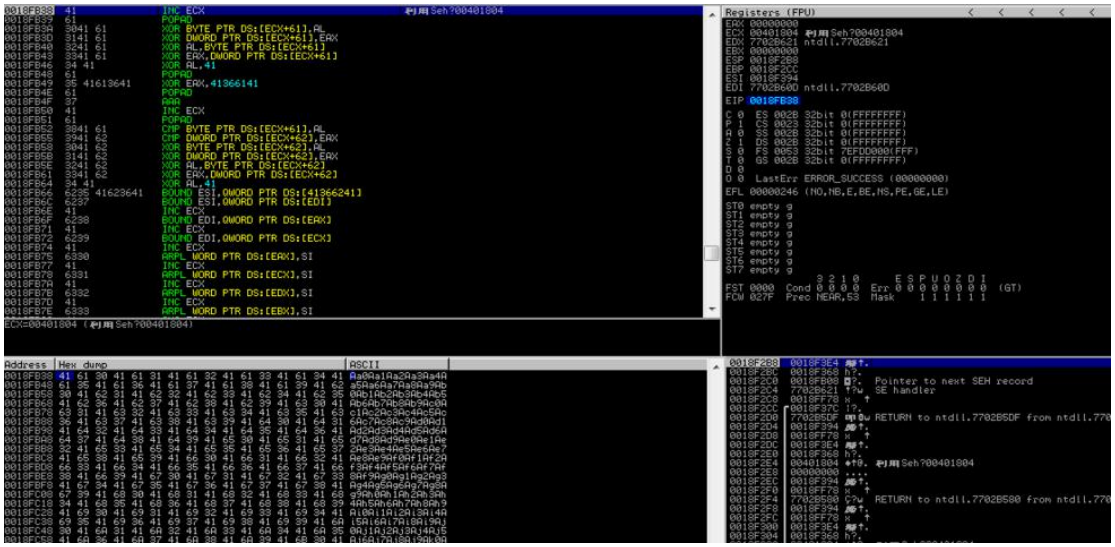


上图是覆盖 Pointer to NextRecord 为 jmp 0018FF80 为向下跳转,单步一步,然后这里需要一个向上跳转的 jmp 这里就用我们文本的第一个字节作为 shellcode 起始位置

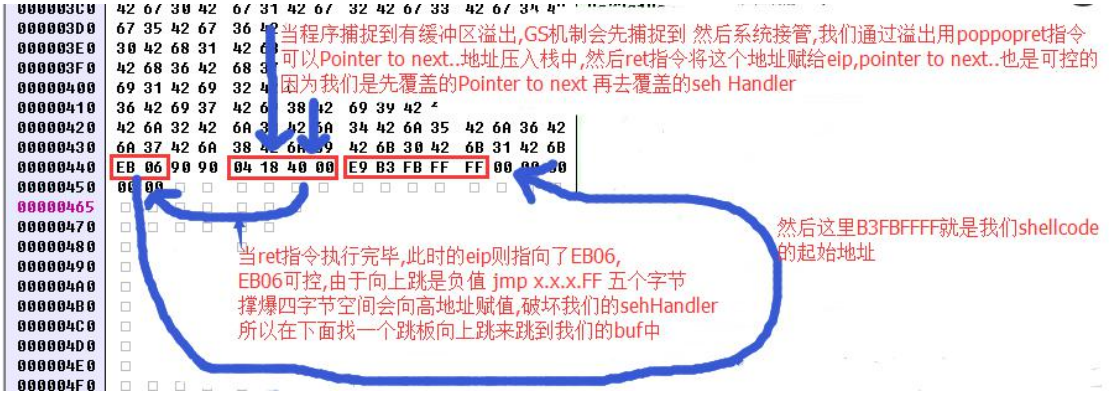
0018FF80 jmp 0018FF80 的二进制是 E9 B3 FB FF



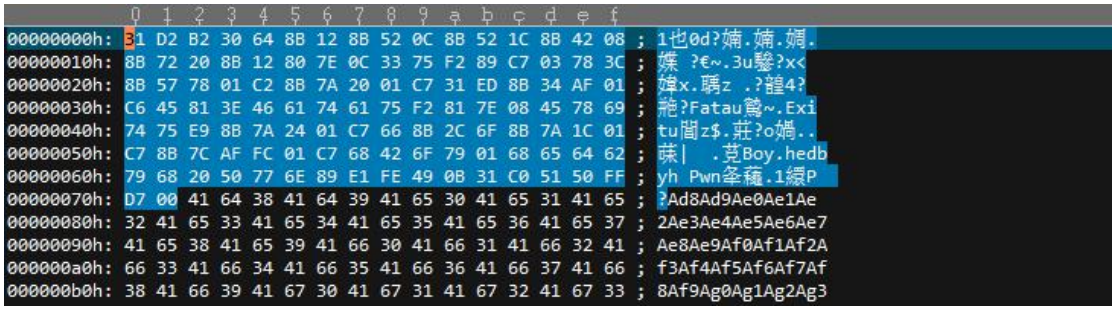
此时 0018FF80 地址处的指令就是 jmp 0018FF80　单步就到了我们的 buf 头了

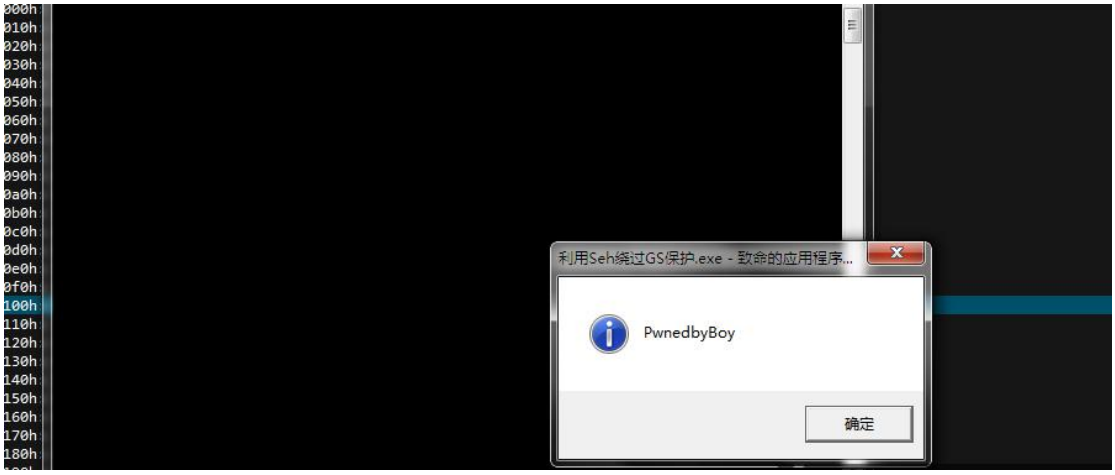


可以看到我们覆盖的数据了，二进制是这样的



接下来我们就在 buf 中扣一段 shellcode 在 0018FF80 这里写指令跳到它的首地址直接用第一个字节的地址　0018FB38





现在我们总结出了一个过程:

1. 找到 SEH 处理函数, 寻找跳板 pop pop ret 来覆盖掉 ntdll 中的 seHandler
2. 构造跳板跳向 shellcode,字节长度问题可以在 seHandler 下方找跳板间接找跳板跳向 shellcode

注意事项:在实验的时候当你在 pop pop ret 下断点的时候,在程序开始会断下,需要跑几次配合 shift+F9 来观察堆栈数据和 eip 指向.